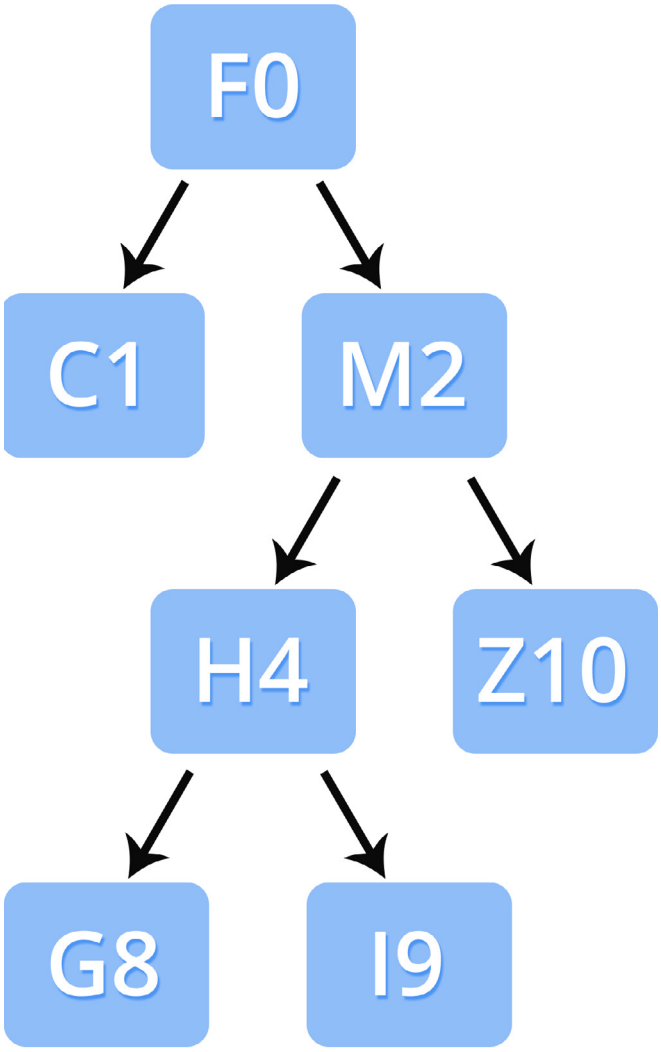
# TP0
# TREE + HEAP = TREAP




**Victor Pires Diniz** - *victorpiresdiniz@dcc.ufmg.br*

AEDS III - 1º Semestre de 2015

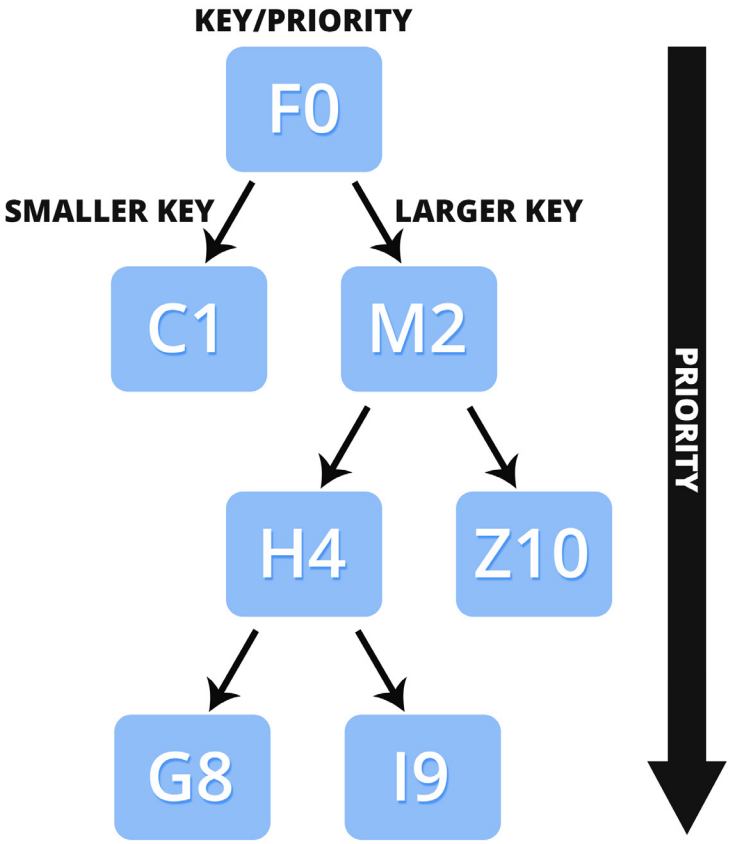
UFMG UNIVERSIDADE FEDERAL DE MINAS GERAIS

# 1 // INTRODUCTION

The search for abstract data types for specific use cases has been an ongoing issue ever since the beginnings of computer science. This has proven not to be an easy task, since there's a large variety of options, each one with a set of benefits and drawbacks. For that reason, it is necessary to correctly evaluate the needs of the project in which it'll be used and the consequences of such a choice.

This work is about an abstract data type called a **treap**. It is a hybrid between a binary search tree and a heap, fulfilling both of these structures' properties in regards to its operations and its organization. Its nodes have two unique key fields, one of which shall be called a *priority* from here on out to avoid confusion.

In the same way as a binary search tree, each node can have up to two child nodes, and they also follow the same rules regarding to their keys: a node's left and right children must have keys smaller and larger than its own, respectively. However, the nodes' priorities must also abide by the heap's rule: the priorities of the children must be larger than the parent's. These organizational rules make it so that if the property values are random, it is extremely likely that the tree will be balanced. This is an advantage over simply using a regular BST, which can easily become unbalanced and unwieldy. It is, of course, possible for a treap to be unbalanced as well, but that's very unlikely. The fact these properties are enforced at the same time also means that treaps, unlike binary search trees or heaps, are unique for a given set of key-priority pairs. This means they can be useful in situations that require this property and also makes it so that equality comparisons are possible.




Lastly and luckily, these conveniences don't take a significant toll on performance in comparison to a regular binary search tree. The basic operations of the treap are relatively simple, compact and elegant: insertion and removal can be done by splitting and merging treaps, which are easily implemented. Searching is done exactly like in a binary search tree. All of this makes the treap an attractive structure for those looking for an ordered data structure for their needs that should remain balanced.

As previously described, the treap is very similar to a binary search tree in its structure. In that sense, its nodes have pointers to two children treap nodes: **left** and **right**. One could include a pointer to the parent node, but it was found unnecessary for this implementation. In addition, there are two integer fields: **key** and **priority**. Key is the actual identifier of the node, whereas priority is used for the heap property.

For this kind of structure, three operations immediately come to mind: insertion of an element, removal of an element and searching for an element. However, these operations are not the main problem to be tackled here, considering the approach that was enforced by this work's specification: these operations have to be implemented through splitting and merging subroutines.

# SPLIT

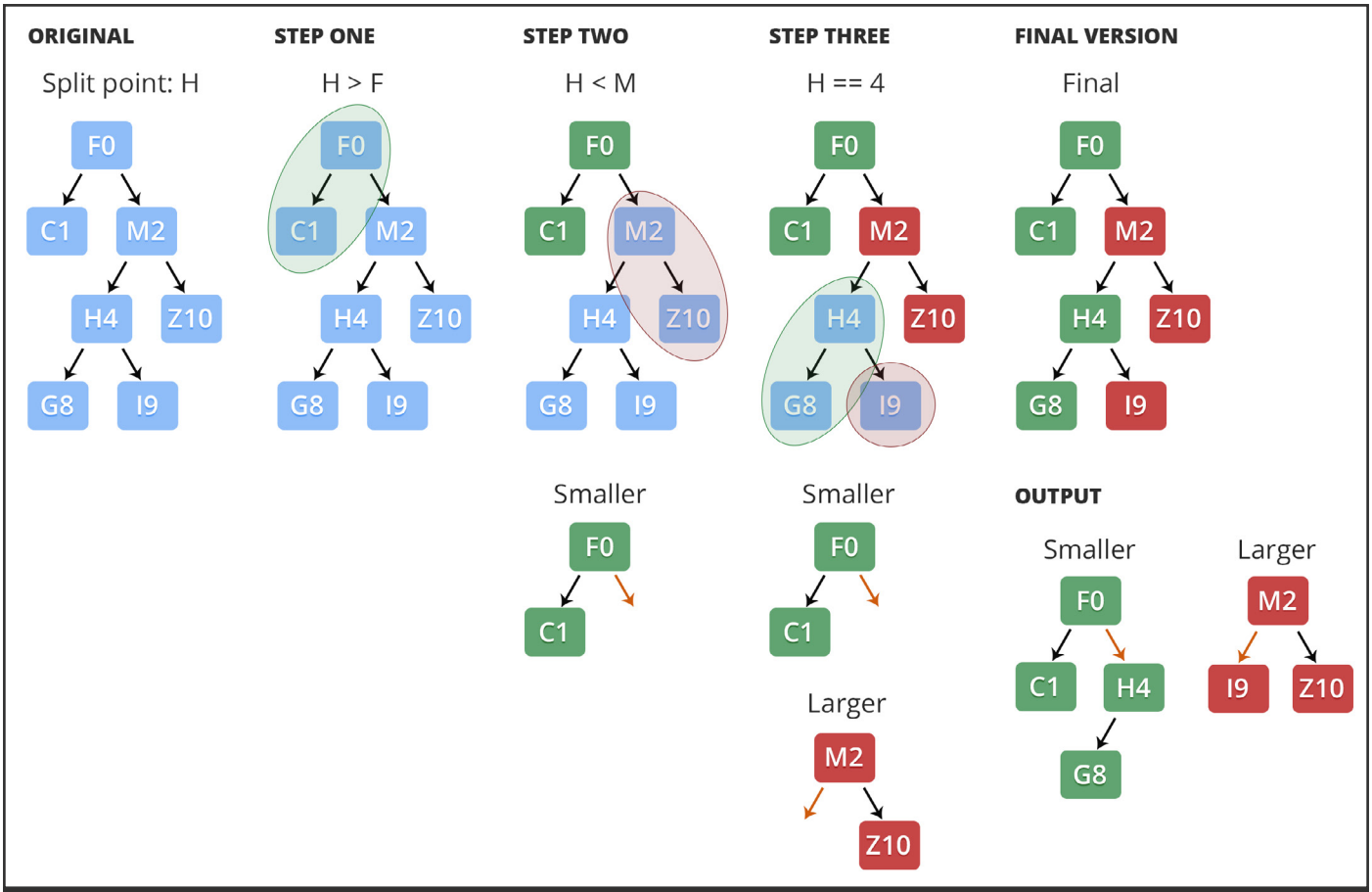
```
void Tr_Split(Treap *source, int split_point);
```

The split operation has no requirements for the treap other than its validity as a treap. It takes only one argument, which is an integer __split point__ that specifies where the treap should be split. Two treaps are to be returned: one with elements whose keys are smaller or equal to the split point and other with the remaining elements, with larger keys. The original treap doesn't need to be preserved intact, so the implementation used simply modifies the existing treap.

The chosen implementation is simple and effective. The pseudocode below describes it:

```
function split(source [treap], split_point [integer]):
    smaller_branch [treap pointer] = &smaller
    larger_branch [treap pointer] = &larger
    smaller [treap] = larger [treap] = NULL
    while source != NULL and source.key != split_point:
        if split_point > source.key:
            *smaller_branch = source
            smaller_branch = &source->right
            source = source->right
        else:
            *larger_branch = source
            larger_branch = &source->left
            source = source->left
    smaller_branch = source
    if source != NULL:
        *larger_branch = source->right
    else:
        *larger_branch = NULL
    return smaller, larger
```

The algorithm basically performs a search for `split_point` and sections the tree depending on the key of the current sub-treap's source. `smaller` and `larger` refer to the treaps whose nodes have smaller or equal and larger keys, respectively. If *split point*'s key is larger than *source*'s, then source and all nodes to its left should be added to the smaller or equal output treap. This means only the right of the source is left for analysis. The same, symetrically, is true if split




*Step-by-step diagram of the split procedure being done to an example treap.*

point's key is smaller or equal to source's. Source is, then, updated to become its right or left child and the process is repeated.

`smaller_branch` and `larger_branch` are pointers used to keep track of where the tree split sections should be "grafted". They're very convenient, because they allow the program to keep track of which one of source's parent's child pointers to update without having to perform extra checks or actually having a parent node pointer in each node.

The loop can be interrupted in two different occasions: it either found an element with split point as its key or it couldn't find anything. Regardless, `smaller_branch` and `larger_branch` need to be updated one last time. If it found the element, then itself and its smaller children should be added to the smaller treap, and its larger children should be added to the larger treap. Otherwise, the branches should be set to null to fix any broken links.

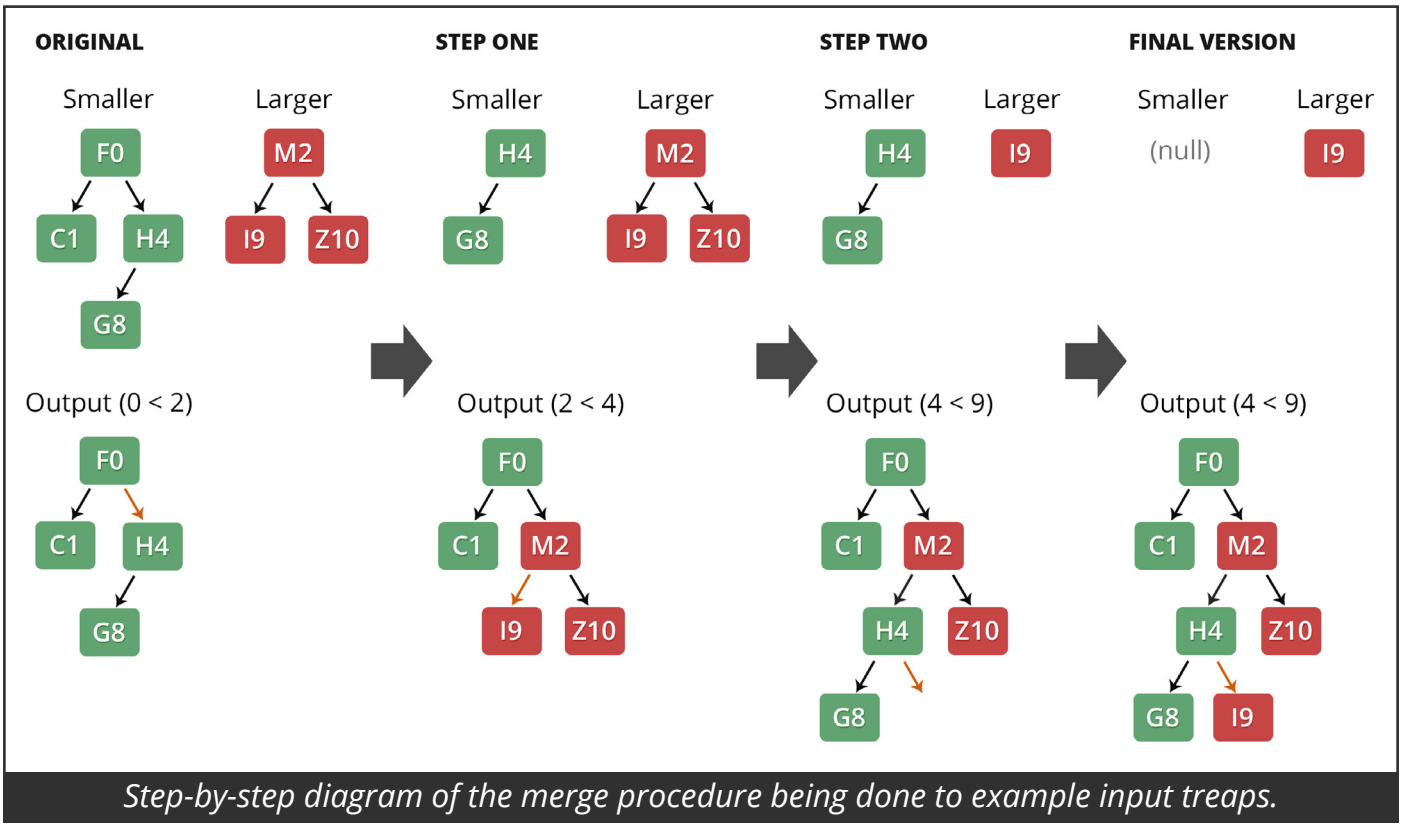
# MERGE

```
Treap *Tr_Merge(Treap *smaller, Treap *larger);
```

This operation can be described as the opposite of splitting: it takes two treaps and joins them into a single structure. However, the elements of the first treap must all be smaller than the elements of the larger treap for it to work properly. Enforcing this property lets the procedure for merging the treaps be compact and efficient.

```
function merge(smaller [treap], larger [treap]):
    root [treap] = NULL
    branch [treap pointer] = &root
    while true:
        if smaller == NULL or larger = NULL:
            if smaller == NULL:
                *branch = larger
            else:
                *branch = smaller
            break
        else if larger.pri < smaller.pri:
            *branch = larger
            branch = &larger->left
            larger = larger->left
        else:
            *branch = smaller
            branch = &smaller->right
            smaller = smaller->right
    return root
```

Rather than comparing keys, what leads the procedure this time is a comparison between the priorities of the two treaps. Whichever one has the smallest priority gets put at the root of the returned treap. Then, similarly to the split procedure, the branch pointer is updated to choose into which child pointer the next treap should be merged. The treap which has just been added to the return treap is updated so as to become its sub-treap in the same direction. This is necessary because parts of the other treap might need to be added in between.




*Step-by-step diagram of the merge procedure being done to example input treaps.*

The logic behind these steps is a little bit more complicated than the one in the split algorithm. Finding out which element to insert in the root of the sub-tree of the current iteration is trivial, but the priority comparison also provides the information needed to figure out which side to branch in the operation. If the smaller treap is the one added to the root, this makes it so that if the larger treap is to be inserted any time in the future, it must be inserted to the right of this root. The opposite is also true: if the larger treap is inserted, then the operation must branch to the left.

Finally, the first comparison of the loop is the stopping point. If one of the treaps is null, there's not much left to do: the program simply inserts the other one in the root and finishes the operation.

# INSERT, REMOVE AND LOCATE

```
Treap *Tr_Insert(Treap *treap, int key, int pri);
Treap *Tr_Remove(Treap *treap, int key);
void Tr_Locate(Treap *treap, int key);
```

The actual operations needed for the interpreter are very simple, once merge and split have been correctly implemented. Insertion can be done by splitting the treap at the key of the new element. Then, merging the smaller treap with the new node, and merging the result with the larger treap.

Removal involves one extra split: the operation consists of splitting the treap at the key of the element to be removed and splitting the smaller treap of that operation once again at the predecessor of the key (key - 1). The larger treap of this procedure will be the node to be removed, if it exists. Then, the treaps can be mended by merging the smaller treap of the second split with the larger treap of the first one.

Finally, locating an element by its key works exactly like searching in a binary search tree and printing the branching directions of each operation. However, if the element is not found, -1 must be printed. This makes it necessary to use a structure to store the directions and choose whether or not to print it after going over the treap, if doing the operation in a single pass is necessary. The implementation chosen simply passes through the treap twice: once to check for existence, and another time to print the directions if the element is actually there.

# AUXILIARY OPERATIONS

```
Treap *Tr_Init(int key, int parent);
void Tr_DeleteNode(Treap *treap);
void Tr_Delete(Treap *treap);
```

There are three functions that haven't been covered in the previous paragraphs. `Tr_Init` initializes a node by allocating memory for it and setting its key-priority pair. `Tr_DeleteNode` is simply a wrapper for `free` from the standard library for treaps, freeing a single node. `Tr_Delete` performs a recursive tree traversal and deletes all nodes in the treap.

Finally, there is also the main function, which is a simple interpreter implemented according to the specification of this problem, which accepts commands for inserting, removing and locating nodes in a treap. It makes testing the library easy and straightforward through an extremely simple syntax.

Before each function is analyzed individually, there's a very important point to be made about the height of a treap. By having the heap properties as well as the binary search tree properties, the treap acts like a randomized binary search tree in regards to its height being O(log n) in most real-life use cases. A way to explain this phenomenon is by taking the priority's influence in a node's positioning similarly to the way the order of insertion affects the organization of a regular BST's elements. If the priorities are properly randomized, the treap behaves just like a randomized binary search tree.

For convenience, in the following analyses, the complexity of each function is analysed in regards to its height. $O(\text{height})$ (or $O(h)$) is, as explained in the previous paragraph, in most situations equivalent to $O(\log n)$, but the worst case scenario still is $O(n)$.

# MERGE

This function has a loop that iterates down the two input treaps (a and b). It loops until the function inserts a leaf node of one of the treaps. This makes one of the treap pointers null and triggers the conditional with the break in it. At the worst case, this loop can go on as many times as the sum of the height of both trees. Therefore, this procedure is $O(\text{a's height} + \text{b's height})$.

# SPLIT

The split procedure also iterates down the input treap, and loops until it finds either the node with split point as its key or a leaf node. In the same way as the merge function, the worst case for this function is $O(h)$.

# INSERT

This function simply calls split once and merge twice to merge the halves of the split procedure with the new element. $O(h) + O(h/2 + 1) + O(h/2 + h/2) = O(h)$.

# REMOVE

The removal operation calls split twice and merge once to join the halves of the previous split call, as well as a constant time operation to delete the node. $O(h) + O(h/2) + O(h/2 + h/2) = O(h)$.

# LOCATE

This function is no more than a binary search tree search algorithm. At worst case, it'll go on until it reaches a leaf node, which means going through as many elements as the height of the treap. Thus, this function's assymptotic complexity is $O(h)$.

# INIT, DELETENODE

Both functions simply perform a series of operations whose performance has no relation with the number of elements in the treap. $O(1)$.

## DELETE

This function traverses through the treap. Having to go through every element performing constant time operations makes the procedure $O(n)$.

# 4 // EXPERIMENTAL ANALYSIS

In order to get data about the actual execution time of this program, a test-making Python script has been made. It creates tests for a given number of random nodes to be inserted ($n$), which have unique keys and priorities that range from 0 to $10^9$. Each node is inserted in random order. Then, a specified number of random locate operations is performed (the elements which are searched for exist in the treap). Finally, the nodes are removed in ascending key order.

The actual benchmarking was accomplished by the `time` Unix command. Four tests were benchmarked, with increasing amounts of nodes and operations:

- 5,000 nodes (5,000 insertions, 5,000 removals, 20 searches);
- 50,000 nodes (50,000 insertions, 50,000 removals, 20 searches);
- 500,000 nodes (500,000 insertions, 500,000 removals);
- 5,000,000 nodes (5,000,000 insertions, 5,000,000 removals);
- 50,000,000 nodes (50,000,000 insertions, 50,000,000 removals);
- 100,000,000 nodes (100,000,000 insertions).

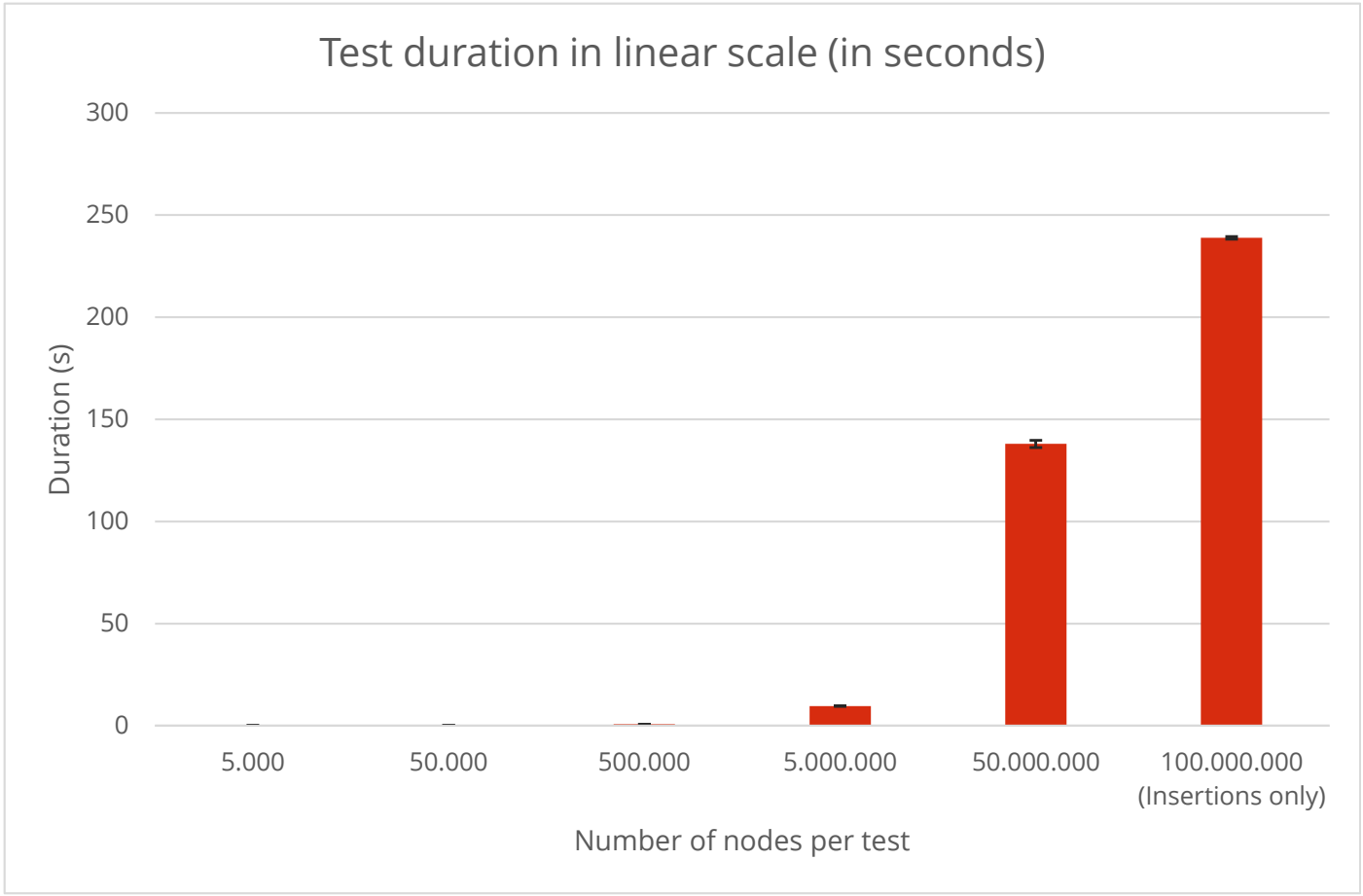
## TESTING ENVIRONMENT AND RELEVANT INFORMATION

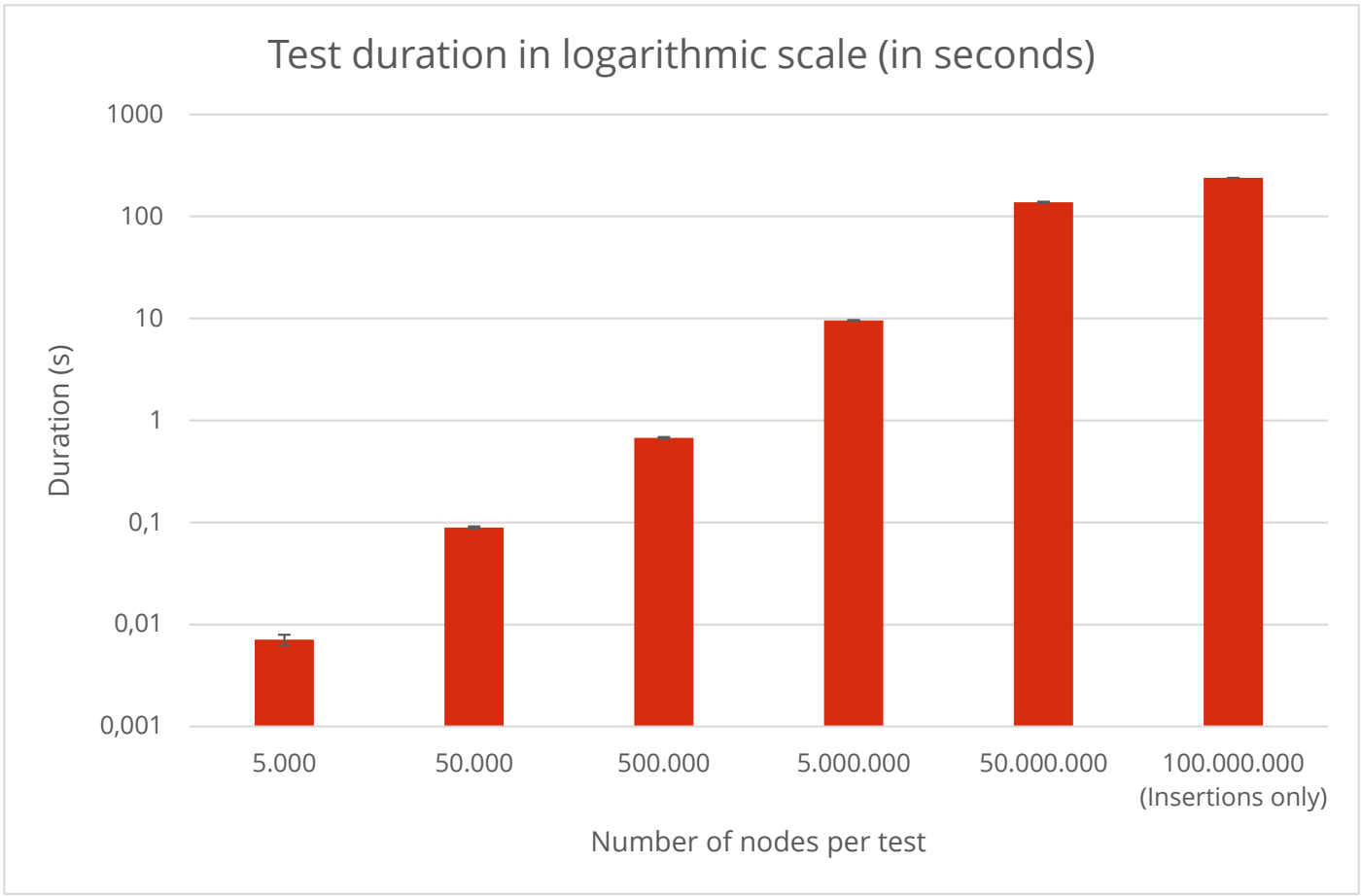
- OS: Arch Linux (x64)
- CPU: Intel Core i7 2600k (3.4GHz)
- RAM: 4 x 4GB DDR3 1,3MHz (16GB total)

The tests were performed ten times each on March 28th, 2015. The following data is an average of the time measured in each of the ten executions.

## RESULTS

- 5,000 nodes: **0,007s;**
- 50,000 nodes: **0.089s**;
- 500,000 nodes: **0.674s**;
- 5,000,000 nodes: **9.572s**;
- 50,000,000 nodes: **137.932s**;
- 100,000,000 nodes (insertions only): **238.842s**.


Test duration in linear scale (in seconds)
300
250
200
150
100
50
0
Duration (s)
5.000
50.000
500.000
5.000.000
50.000.000
100.000.000
(Insertions only)
Number of nodes per test


Test duration in logarithmic scale (in seconds)
1000
100
10
1
0,1
0,01
0,001
Duration (s)
5.000
50.000
500.000
5.000.000
50.000.000
100.000.000
(Insertions only)
Number of nodes per test

This data shows that the program runs in feasible time for all planned input sizes. In particular, the two last examples both feature the maximum amount of instructions allowed, and the last one has the maximum amount of nodes possible, by only having insertion operations. This is possible because, in the end of the program's execution, the treap is deleted in its entirety regardless of removal operations.

The yielded data also matches the assymptotic complexity analysis previously observed, since the program runs n operations that perform at $O(\log n)$, which results in a $O(n \log n)$ complexity for the entire execution. Increasing the number of nodes and operations tenfold is expected to increase the program's execution time by roughly $10 * \log 10 * k$ times, $k$ being a positive, real constant that depends on many disregarded factors. The logarithmic scale graph also reinforces this observation, because the number of nodes and operations is being increased exponentially, and, yet, the graph appears to be linear.

# 5 // CONCLUSION

In this project, a treap data structure has been properly implemented. Its basic operations can be done very simply by calling the merge and join procedures. These procedures were assymptotically analysed and proven to be able to be performed in $O(\log n)$. To further confirm that information, benchmarks with large data sets have been executed and resulted in data that corresponded to the expected performance.

Further improvements to this can be done, depending to the operations that are called the most and would gain the most from having better performance. A common improvement done to the treap is to reduce the priority of an element which is accessed frequently, bringing it closer to the root and reducing its access time, in conformity with the principle of temporal locality of reference. These kinds of enhancements have downsides, however, and making such features could make this program much more complex, which would jeopardize the beautiful simplicity of this implementation.